Homework 6: Scheduling

Course: CS 221 Spring 2019
Name: Bryan Yaggi

What courses should you take in a given quarter? Answering this question requires balancing your interests, satisfying prerequisite chains, graduation requirements, availability of courses; this can be a complex tedious process. In this assignment, you will write a program that does automatic course scheduling for you based on your preferences and constraints. The program will cast the course scheduling problem (CSP) as a constraint satisfaction problem (CSP) and then use backtracking search to solve that CSP to give you your optimal course schedule.

You will first get yourself familiar with the basics of CSPs in Problem 0. In Problem 1, you will implement two of the three heuristics you learned from the lectures that will make CSP solving much faster. In Problem 2, you will add a helper function to reduce n-ary factors to unary and binary factors. Lastly, in Problem 3, you will create the course scheduling CSP and solve it using the code from previous parts.

Problem 0: CSP Basics

(a) Let's create a CSP. Suppose you have n light bulbs, where each light bulb $i = 1, \ldots, n$ is initially off. You also have m buttons which control the lights. For each button $j = 1, \ldots, m$, we know the subset $T_j \subseteq \{1, \ldots, n\}$ of light bulbs that it controls. When button j is pressed, it toggles the state of each light bulb in T_j (For example, if $3 \in T_j$ and light bulb 3 is off, then after the button is pressed, light bulb 3 will be on, and vice versa).

Your goal is to turn on all the light bulbs by pressing a subset of the buttons. Construct a CSP to solve this problem. Your CSP should have m variables and n constraints. For this problem only, you can use n-ary constraints. Describe your CSP precisely and concisely. You need to specify the variables with their domain, and the constraints with their scope and expression. Make sure to include T_j in your answer.

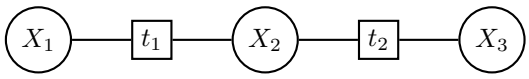
There are m variables, one for each button. The domain consists of 1 or 0, indicating whether the button is pressed or not.

$$X = (X_1, \ldots, X_m), where\ X_j \in \{0, 1\}$$

There are n factors, one for each light. The factor will be 1 if the number of pressed buttons controlling the light is odd.

$$f_i = \mathbb{1}\left[\left(\sum_{j:i \in T_j}^{m} X_j\right) \mod 2\right]$$

(b) Let's consider a simple CSP with 3 variables and 2 binary factors:



where $X_1, X_2, X_3 \in \{0, 1\}$ and t_1, t_2 are XOR functions (that is $t_1(X) = X_1 \oplus X_2$ and $t_2(X) = X_2 \oplus X_3$).

i. How many consistent assignments are there for this CSP?

X_1	X_2	X_3	t_1	t_2	$weight$
0	0	0	0	0	0
1	0	0	1	0	0
0	1	0	1	1	1
1	1	0	0	1	0
0	0	1	0	1	0
1	0	1	1	1	1
0	1	1	1	0	0
1	1	1	0	0	0

There are 2 consistent assignments: $x = \{X_1 = 0, X_2 = 1, X_3 = 0\}$ and $x = \{X_1 = 1, X_2 = 0, X_3 = 1\}$.

ii. To see why variable ordering is important, let's use backtracking search to solve the CSP without using any heuristics (MCV, LCV, AC-3) or lookahead. How many times will `backtrack()` be called to get all consistent assignments if we use the fixed ordering X_1, X_3, X_2? Draw the call stack for `backtrack()`. (You should use the Backtrack algorithm from the slides. The initial arguments are $x = \emptyset, w = 1$, and the original Domain.)

In the code, this number will be stored in `BacktrackingSearch.numOperations`.

Use domain ordering $Domain = \{0, 1\}$. The top of the stack is on the bottom.

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0, X_3 = 0\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0, X_3 = 0, X_2 = 1\}$`, `$w = 1$`)` *consistent assignment found*

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0, X_3 = 1\}$`, `$w = 1$`)`

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1, X_3 = 0\}$`, `$w = 1$`)`

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1, X_3 = 1\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1, X_3 = 1, X_2 = 0\}$`, `$w = 1$`)` *consistent assignment found*

`backtrack()` is called 9 times.

iii. To see why lookahead can be useful, let's do it again with the ordering X_1, X_3, X_2 and AC-3. How many times will Backtrack be called to get all consistent assignments? Draw the call stack for `backtrack()`.

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0\}$`, `$w = 1$`)` $Domain_3 = \{0\}$ $Domain_2 = \{1\}$
`backtrack(`$x = \{X_1 = 0, X_3 = 0\}$`, `$w = 1$`)` $Domain_2 = \{1\}$
`backtrack(`$x = \{X_1 = 0, X_3 = 0, X_2 = 1\}$`, `$w = 1$`)` *consistent assignment found*

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1\}$`, `$w = 1$`)` $Domain_3 = \{1\}$ $Domain_2 = \{0\}$
`backtrack(`$x = \{X_1 = 1, X_3 = 1\}$`, `$w = 1$`)` $Domain_2 = \{0\}$
`backtrack(`$x = \{X_1 = 1, X_3 = 1, X_2 = 0\}$`, `$w = 1$`)` *consistent assignment found*

`backtrack()` is called 7 times.

(c) coding

Problem 1: CSP Solving

(a) coding

(b) coding

(c) coding

Problem 2: Handling N-ary Factors

So far, our CSP solver only handles unary and binary factors, but for course scheduling (and really any non-trivial application), we would like to define factors that involve more than two variables. It would be nice if we could have a general way of reducing n-ary constraint to unary and binary constraints. In this problem, we will do exactly that for two types of n-ary constraints.

Suppose we have boolean variables X_1, X_2, X_3, where X_i represents whether the i-th course is taken. Suppose we want to enforce the constraint that $Y = X_1 \vee X_2 \vee X_3$, that is, Y is a boolean representing whether at least one course has been taken. For reference, in `util.py`, the function `get_or_variable()` does such a reduction. It takes in a list of variables and a target value, and returns a boolean variable with domain $[True, False]$ whose value is constrained to the condition of having at least one of the variables assigned to the target value. For example, we would call `get_or_variable()` with arguments $(X_1, X_2, X_3, True)$, which would return a new (auxiliary) variable X_4, and then add another constraint $[X_4 = True]$.

The second type of n-ary factors are constraints on the sum over n variables. You are going to implement reduction of this type but let's first look at a simpler problem to get started:

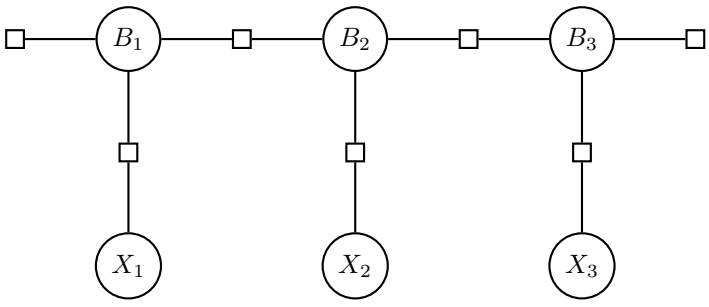
(a) Suppose we have a CSP with three variables X_1, X_2, X_3 with the same domain $\{0, 1, 2\}$ and a ternary constraint $[X_1 + X_2 + X_3 \leq K]$. How can we reduce this CSP to one with only unary and/or binary constraints? Explain what auxiliary variables we need to introduce, what their domains are, what unary/binary factors you'll add, and why your scheme works. Add a graph if you think that'll better explain your scheme.

Define a new variables A_i, B_i.

$$\begin{aligned} A_i &= A_{i-1} + X_i, A_0 = 0 \\ B_i &= (A_{i-1}, A_i) \\ Domain(B_i) &= \{(x, y) : 0 \leq x \leq 6, 0 \leq y \leq 6, x \in \mathbb{Z}, y \in \mathbb{Z}\} \end{aligned}$$

Factors:

- $\mathbb{1}[B_1[0] = 0]$
- $\mathbb{1}[B_3[1] \leq K]$
- $\mathbb{1}[B_i[0] = B_{i-1}[1]]$
- $\mathbb{1}[B_i[1] = B_{i-1} + X_i]$



(b) coding

Problem 3: Course Scheduling

In this problem, we will apply your weighted CSP solver to the problem of course scheduling. We have scraped a subset of courses that are offered from Stanford's Bulletin. For each course in this dataset, we have information on which quarters it is offered, the prerequisites (which may not be fully accurate due to ambiguity in the listing), and the range of units allowed. You can take a look at all the courses in `courses.json`. Please refer to `util.Course` and `util.CourseBulletin` for more information.

(a) coding

(b) coding

(c) Now try to use the course scheduler for the winter and spring (and next year if applicable). Create your own `profile.txt` and then run the course scheduler:

`python run_p3.py profile.txt`

You might want to turn on the appropriate heuristic flags to speed up the computation. Does it produce a reasonable course schedule? Please include your `profile.txt` and the best schedule in your writeup; we're curious how it worked out for you!

```
profile.txt
-----------
minUnits 3
maxUnits 5
taken STATS116
taken CS106A
taken CS106B
taken CS103
taken CS107
taken CS109
taken CS110
taken CS221
request CS161 weight 2
request CS231N weight 2
request CS229
request CS230
request CS234
-----------
```

Resulting best schedule:

Quarter	Units	Course
Sum2019	5	CS161
Aut2019	4	CS229
Win2020	3	CS234
Spr2020	4	CS231N

Note: I added CS230, CS231N, and CS234 to `courses.json`.

Problem 4: Weighted CSPs with Notable Patterns (Extra Credit)

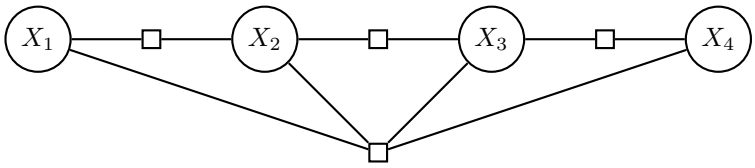
Want more challenges about CSP? Here we go. :D

Suppose we have a weighted CSP with variables $X_1, \ldots, X_n$ with domains $Domain_i = 1, \ldots, K$. We have a set of basic factors which depend only on adjacent pairs of variables in the same way: there is some function g such that $f_i(x) = g(x_i, x_{i+1})$ for $i = 1, \ldots, n1$. In addition, we have a small set of notable patterns P, where each $p \in P$ is a sequence of elements from the domain.

Let n_p be the number of times that p occurs in an assignment $x = (x_1, \ldots, x_n)$ as a consecutive sequence. Define the weight of an assignment x to be $\prod_{i=1}^{n1} f_i(x) \prod_{p \in P} \gamma^{n_p}$. Intuitively, we multiply the weight by γ every time a notable pattern appears.

For example, suppose $n = 4$, $\gamma = 7$, $g(a, b) = 5[a = b] + 1[a \neq b]$ and $P = \{[1, 3, 3], [1, 2, 3]\}$. Then the assignment $x = [1, 3, 3, 2]$ has weight $(1 \cdot 5 \cdot 1) \cdot (71 \cdot 70) = 35$.

(a) If we were to include the notable patterns as factors into the CSP, what would be the worst case treewidth? (You can assume each p has a maximum length of n.)



Worst-case tree width is $n - 1$.

(b) The treewidth doesn't really tell us the true complexity of the problem. Devise an efficient algorithm to compute the maximum weight assignment. You need to describe your algorithm in enough detail but don't need to implement it. Analyze your algorithm's time and space complexities. You'll get points only if your algorithm is much better than the naive solution.